Implementing a Deep Learning Library from Scratch

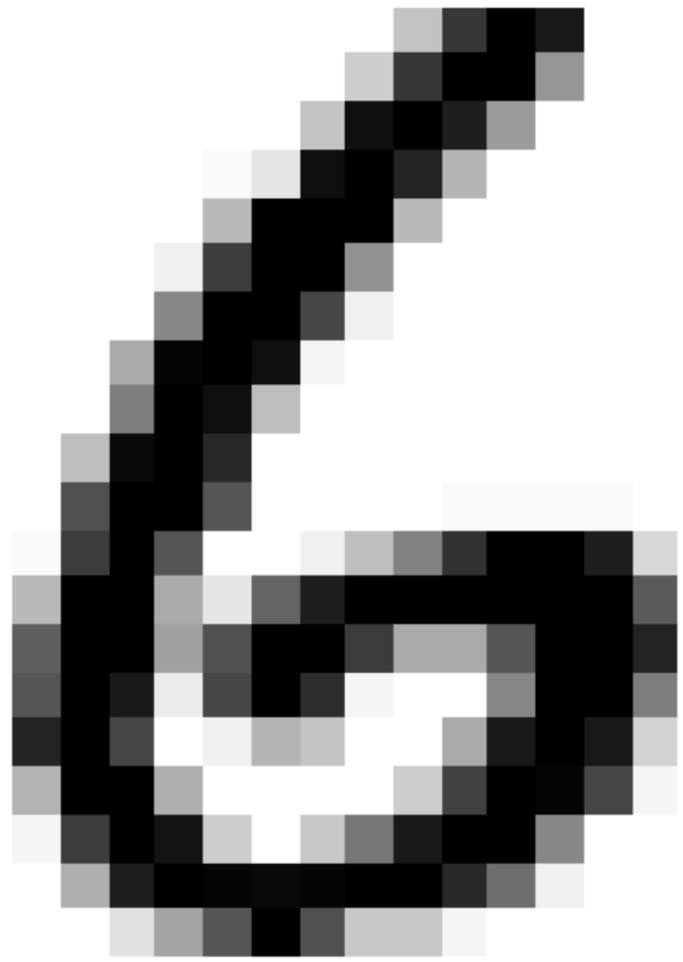
Eric Allen

The MNIST Dataset

- A collection of 70,000 labelled handwritten digits especially designed for use with machine learning experiments
- About 52 MB
- Collected from high school students and Census Bureau employees
- A training set of 60,000 examples
- A test set of another 10,000 examples

MNIST Examples

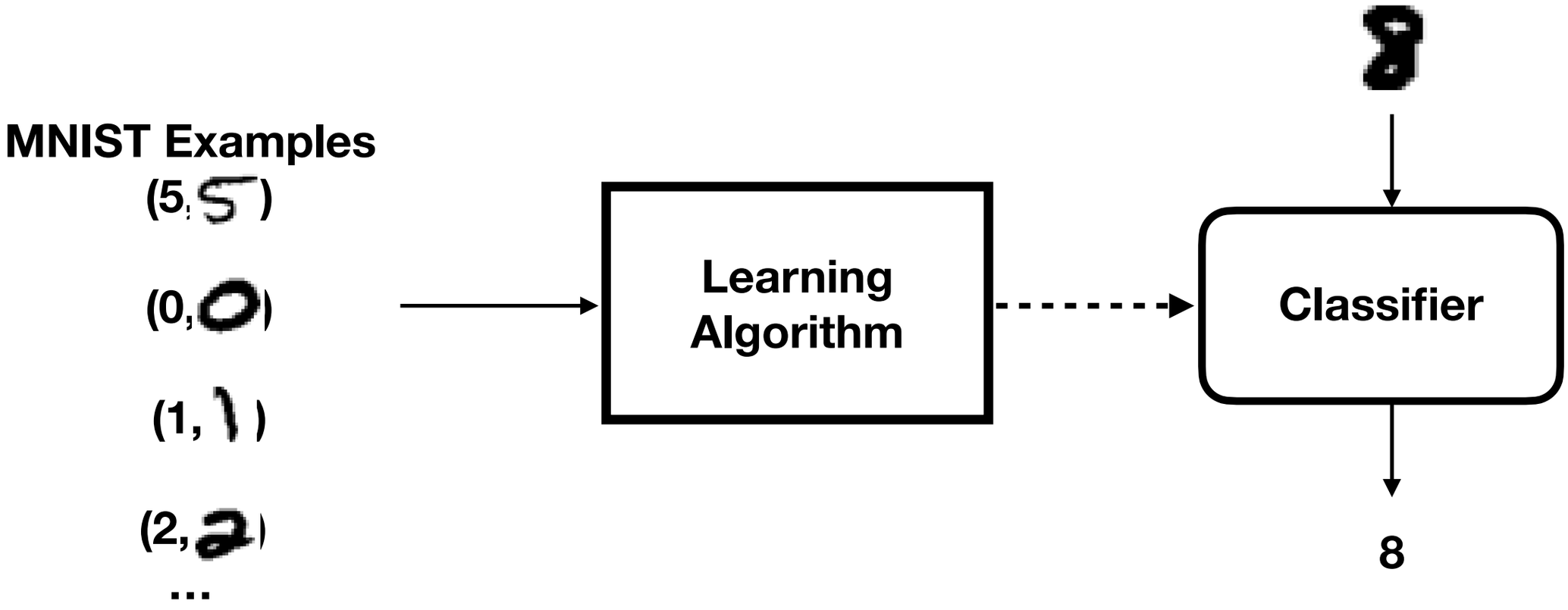
- Each MNIST example consists of:
  - A grayscale image of the digit
  - 28x28 pixels
  - One byte per pixel
  - A one byte label indicating which digit the image denotes (0-9)



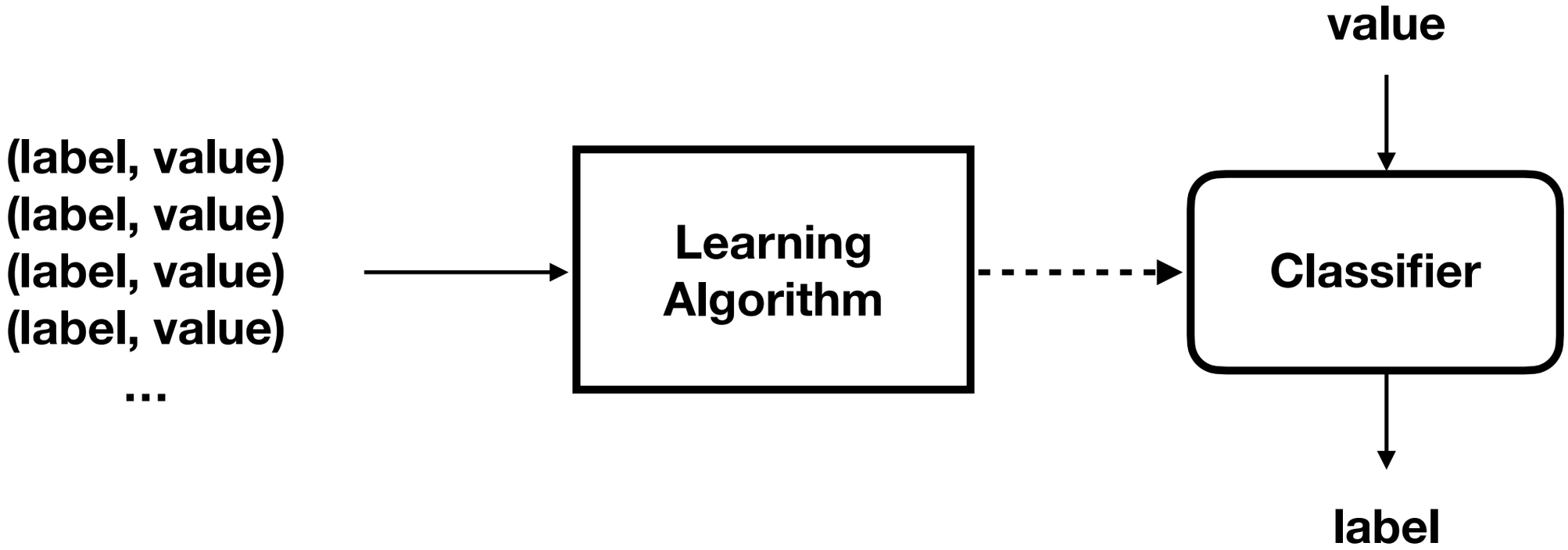
Machine Learning 101

- *Supervised* machine learning algorithms take as input a collection of training examples
- Each training example consists of a value and a label
- The algorithm *learns* from these examples and produces a *classifier*
- Given a new value, the classifier predicts a label for the value, based on what it learned

Machine Learning 101



Machine Learning 101



What Kinds of Labels Can We Provide the Algorithm?

- Typically, labels are values in some enumeration:
- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
- January-December
- ★,★★,★★★,★★★★,★★★★★

What Kinds of Labels Can We Provide the Algorithm?

- There must be at least two possible labels
- Some algorithms only work with two labels
- In MNIST, we have 10 possible labels (0-9)

What Kinds of Example Values Can We Provide the Algorithm?

- Usually, values are encoded as high dimensional vectors of numeric quantities

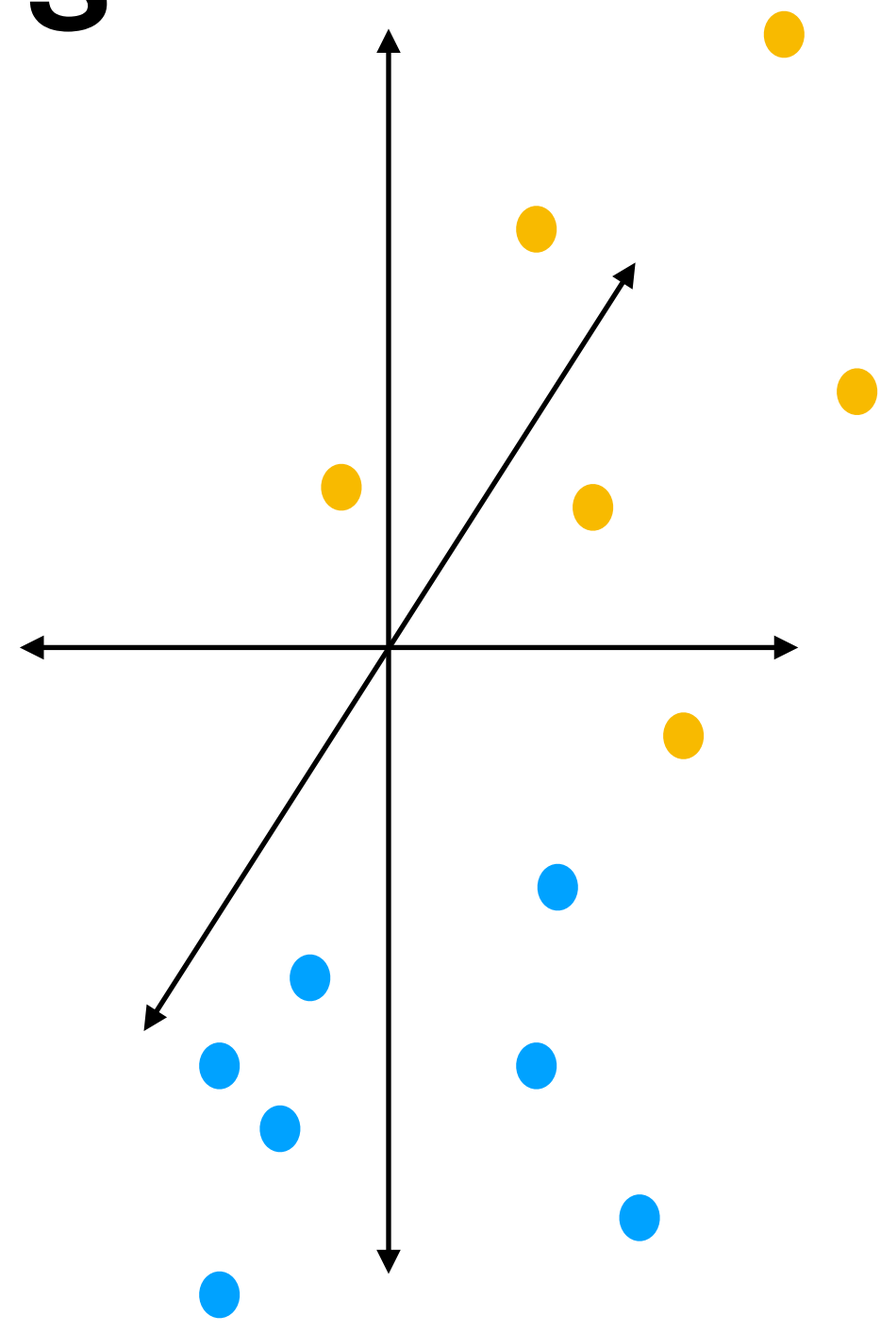
- MNIST examples can be thought of as 784 dimensional vectors

- Each pixel can be thought of as denoting a grayscale value between 0 and 1

Encoding Example Values as Vectors

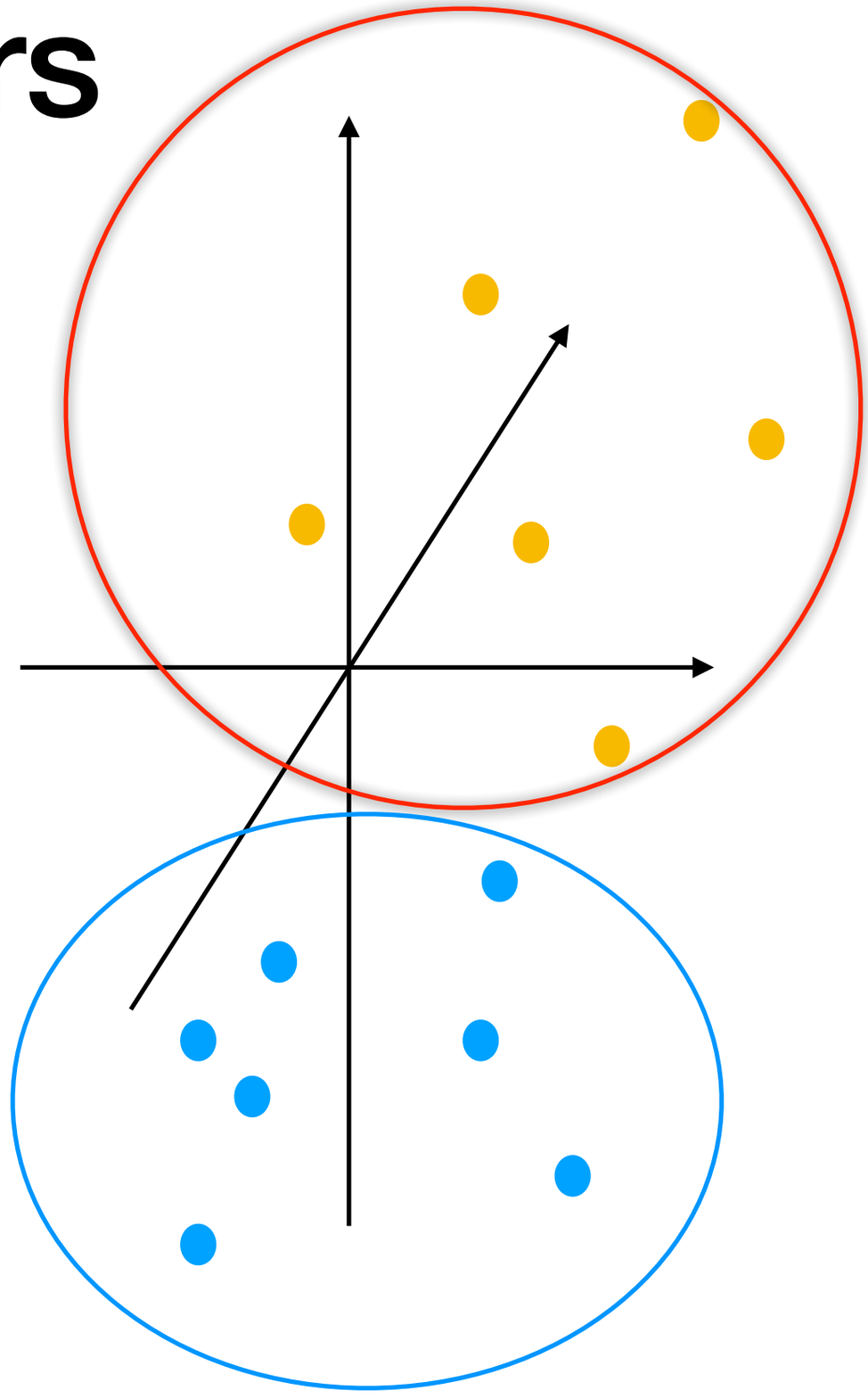
- We can think of each training example as a point in a vector space

- Labels “color” these points



Encoding Example Values as Vectors

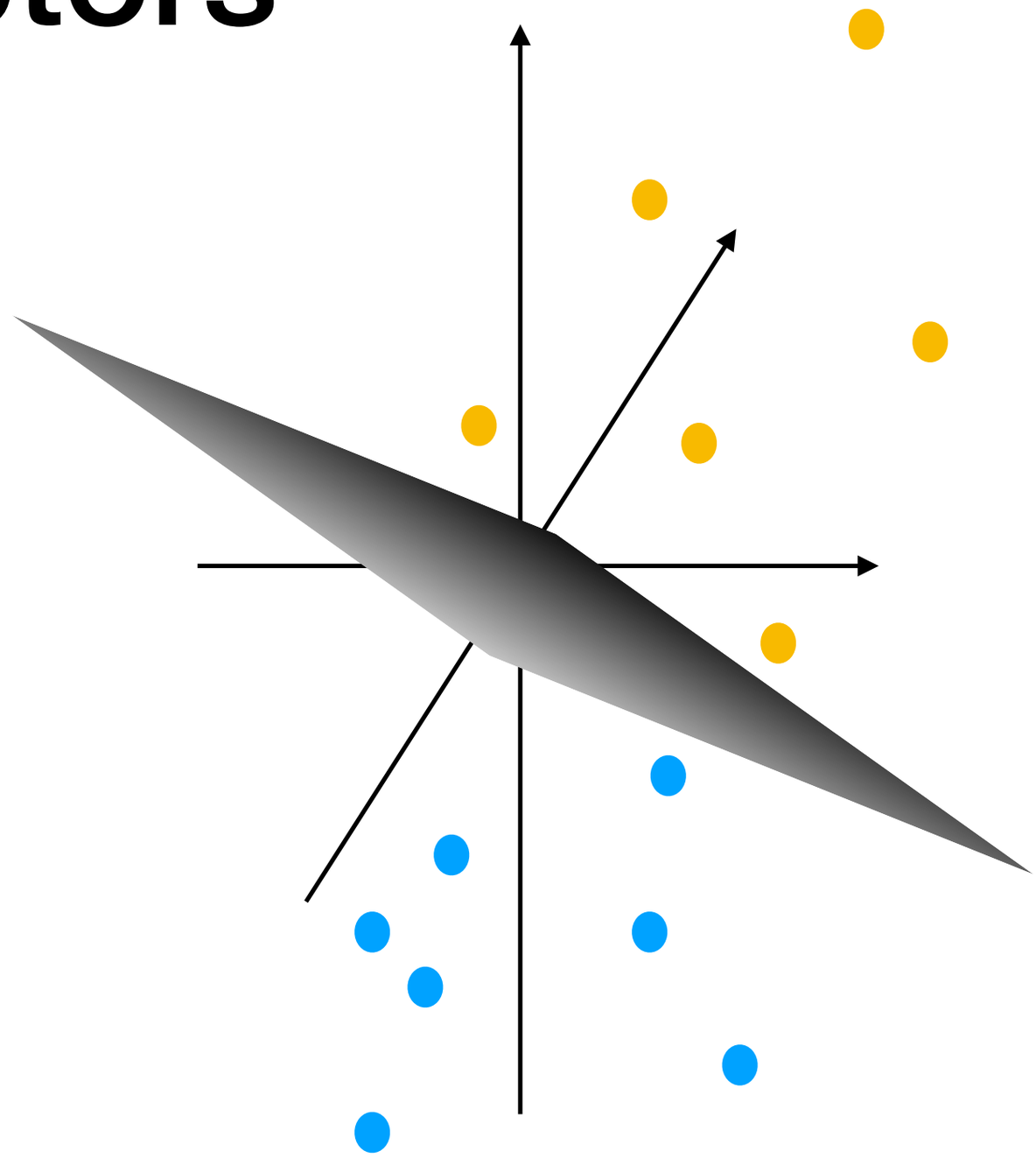
- Then building a machine learning classifier simply involves finding a way of assigning a color to any given point in our vector space



Encoding Example Values as Vectors

- If we are lucky, we can find straight lines (or planes, or hyperplanes) that separate our colored regions

- Then we can use what is called a *linear classifier*

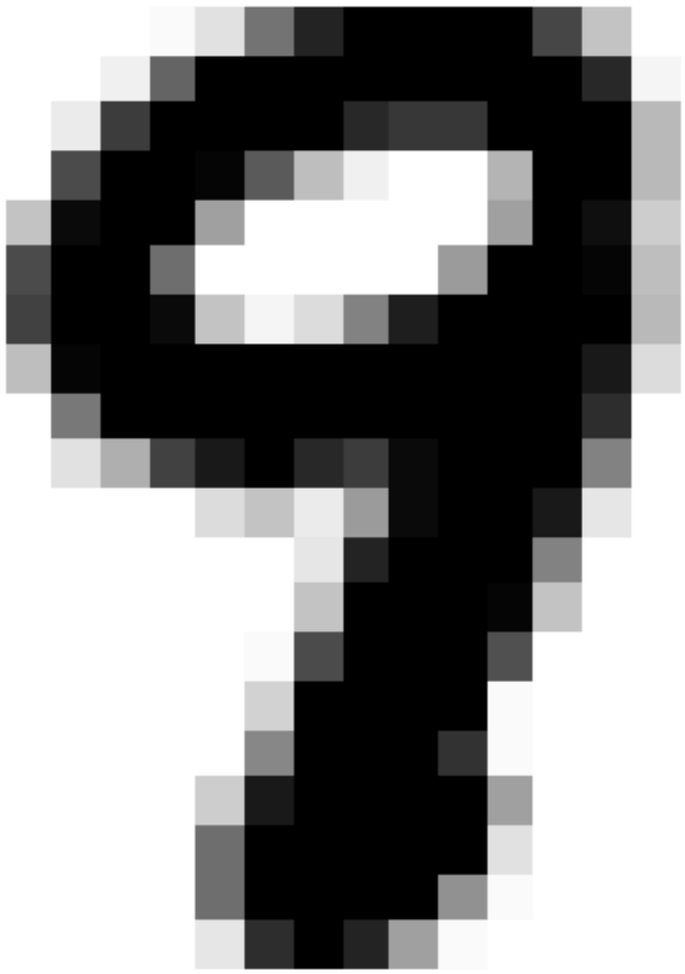


Linear Classifiers

- If we can linearly separate example values, then we can think of each dimension as contributing independently to the answer as to which label applies

- Then we need only learn how strongly to weigh the contribution of each dimension when deciding on a label

Linear Classifiers Are Not a Good Fit for MNIST



Artificial Neural Networks

- Loosely inspired by the structure of biological brains
- A layer of *input neurons* takes values from our training examples
- A *hidden layer* of neurons is connected by weighted edges (synapses) to the neurons on the input layer
- Finally, an *output layer* of neurons is connected to the last hidden layer and provides an answer as to how to label the input example

Artificial Neural Networks

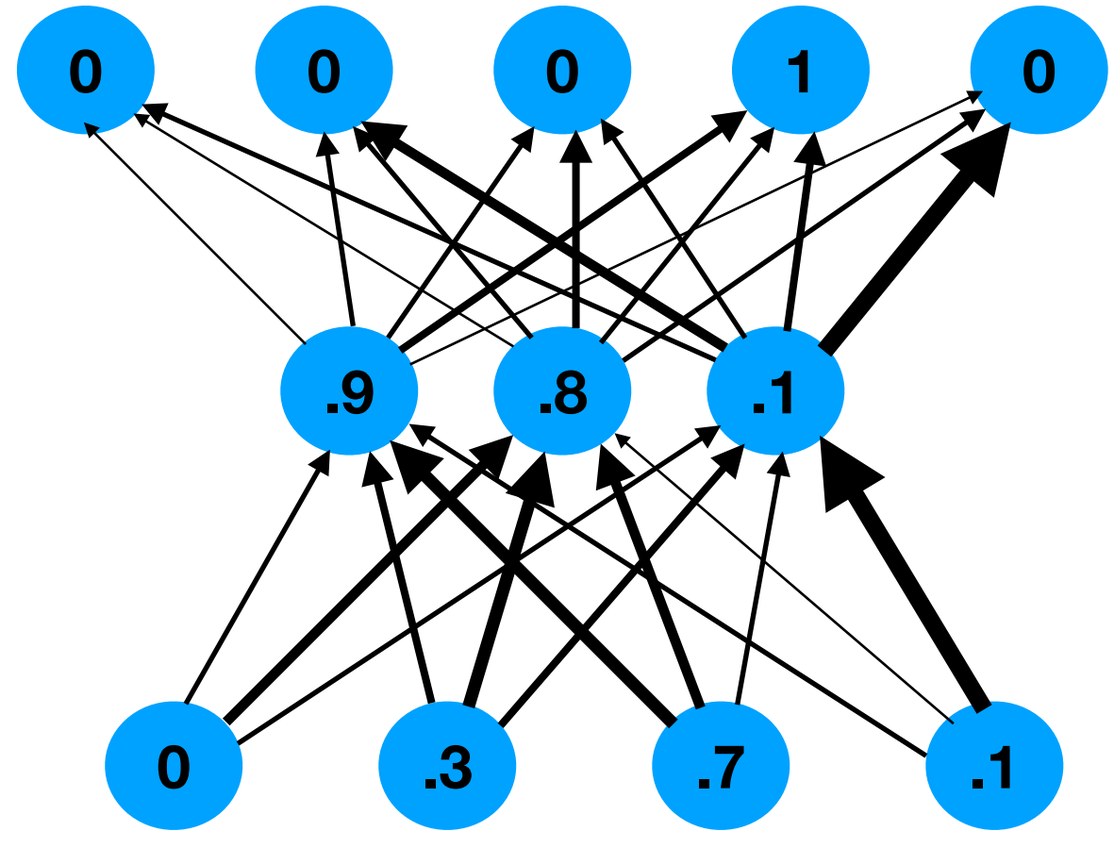
- We can include additional hidden layers

- Each layer is connected to neurons in the previous layer

An Example Neural Network

- Values of input neurons at bottom are determined by an example value

- Values of hidden and output neurons are determined by
  - The values of the neurons in the previous layer
  - The strengths of the synapses



Computing the value of a neuron

- To calculate the value of a neuron *n:*
  - First, calculate the sum *s* consisting of the value of each neuron *m* in the previous layer multiplied by the weight of the synapse connecting *m* to *n*

Computing the value of a neuron

So far, this is just matrix multiplication: Think of the previous layer as a vector and the synapses as a matrix of weights

- To calculate the value of a neuron *n:*
  - First, calculate the sum *s* consisting of the value of each neuron *m* in the previous layer multiplied by the weight of the synapse connecting *m* to *n*

Computing the value of a neuron

- To calculate the value of a neuron *n:*
  - First, calculate the sum *s* consisting of the value of each neuron *m* in the previous layer multiplied by the weight of the synapse connecting *m* to *n*
  - Apply a *threshold function* to this sum
  - This better simulates the behavior of biological neurons

How Do Neural Networks Learn?

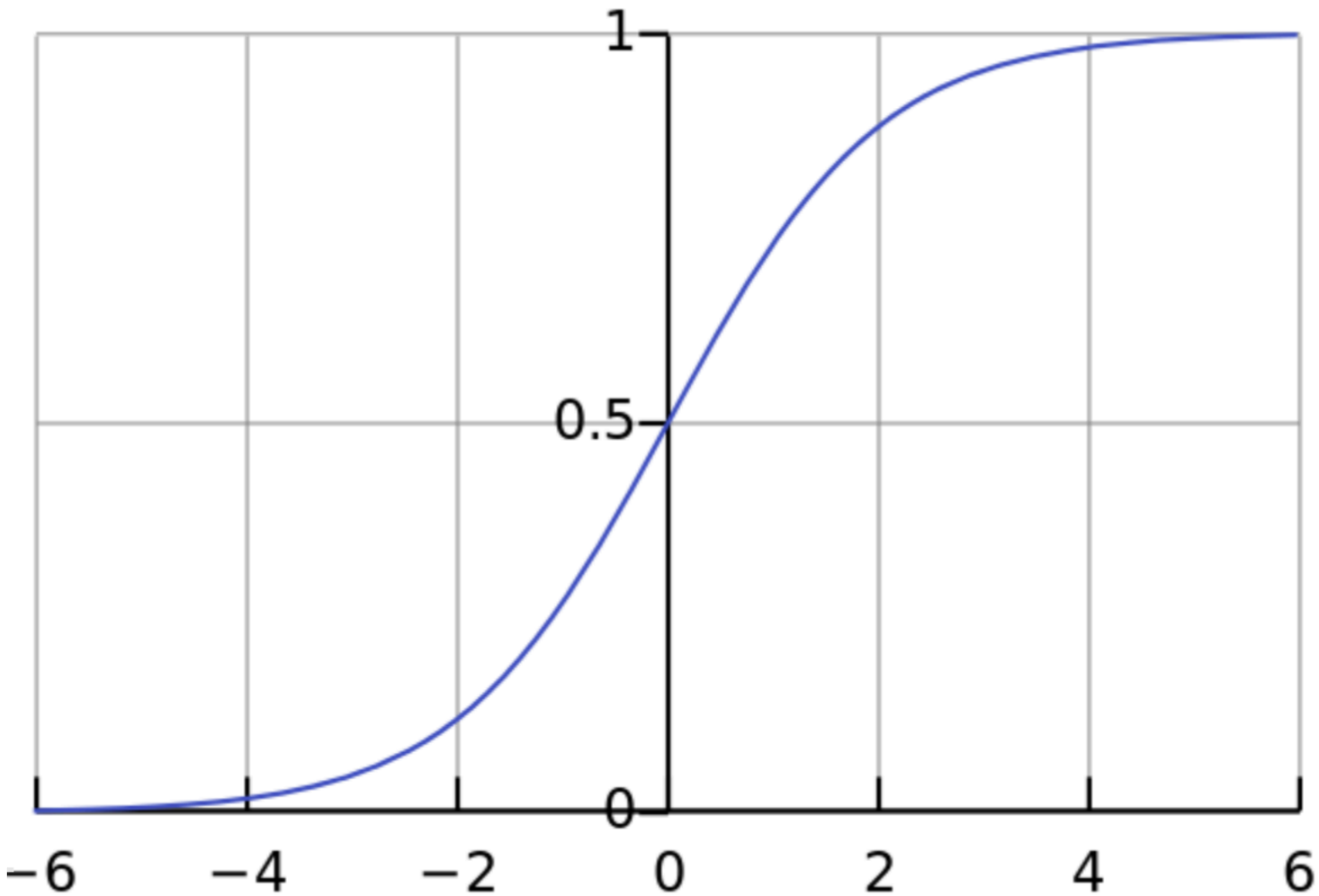
- *Learning*, for a neural network, involves finding weights for the synapses so that the network as a whole maps inputs to the appropriate outputs

A Simple Threshold Function

- 0 if *s* < *0.5*
- 1 if *s* >= *0.5*
- *(This turns out not to be a very good threshold function)*

A Classic Threshold Function: The Logistic Curve

$$S(x) = \frac{1}{1 + e^{-x}}$$



A Classic Threshold Function: The Logistic Curve

$$S(x) = \frac{1}{1 + e^{-x}}$$

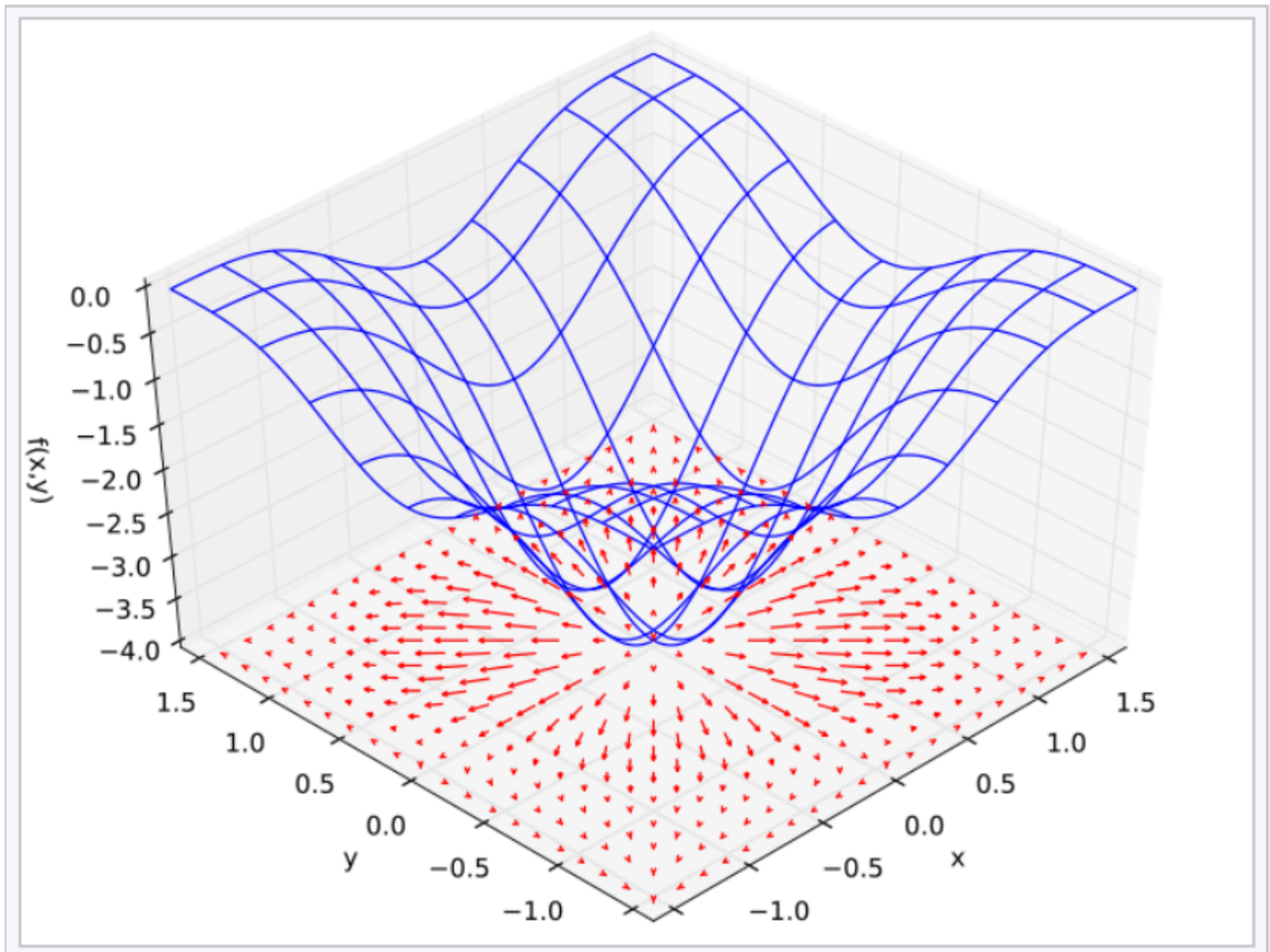
- This has the advantage of being differentiable

- *Who cares?*

Learning with Back Propagation

- Each time the network sees a training example:
  - Feed the input through the network according to the current weights of the synapses
  - Calculate the *distance* between the actual output layer (vector) and the desired output layer (vector)
  - Use the derivative of the function from the previous hidden layer to the output layer to compute a *gradient*

Gradient Descent



The gradient of the function $f(x,y) = -(\cos^2 x + \cos^2 y)^2$ depicted as a projected vector field on the bottom plane.

Back Propagation: Learning Through Stochastic Gradient Descent

- Each time the network sees a training example:
  - …
  - Use the derivative of the function from the last hidden layer to the output layer to compute a *gradient*
  - Similarly, compute the gradient for the last hidden layer, and so on, until we reach the input layer

Back Propagation: Learning Through Stochastic Gradient Descent

- Each time the network sees a training example:
- …
- Now use the computed gradient to update the weights on all layers

Back Propagation: Learning Through Stochastic Gradient Descent

- Continue training until either:
  - A desired performance threshold is reached
  - Or we run out of some resource (such as time)
- It is common to run the same example set through the network many times
  - Each run of the complete example set is referred to as an *epoch*

Implementing a Neural Network Library

- There are many libraries available for neural networks and machine learning:
  - TensorFlow
  - Theano
  - Torch
- Why implement one from scratch?

“Programming properly should be regarded as an activity by which the programmers form or achieve a certain kind of insight, a theory, of the matters at hand.”

–Peter Naur

Experience in Rolling a Neural Network Library From Scratch

- Scala/JVM handles the MNIST dataset well
    - Racket does not
- Debugging and unit testing is interesting
    - Learning even simple functions with simple networks can take many epochs
    - This makes it difficult to quickly ensure there are not bugs in the formulas

Experience in Rolling a Neural Network Library From Scratch

- Debugging and unit testing a neural network is interesting
- Used a simple network that learns xor for testing
- Learning xor took 2000 epochs!

Experience in Rolling a Neural Network Library From Scratch

- Initial synapse weights should be small
  - (-0.1, 0.1) works well

Training and Testing on MNIST

- Initial training set divided into 54,000 training instances and 6000 validation instances

- Have not yet looked at the MNIST test set

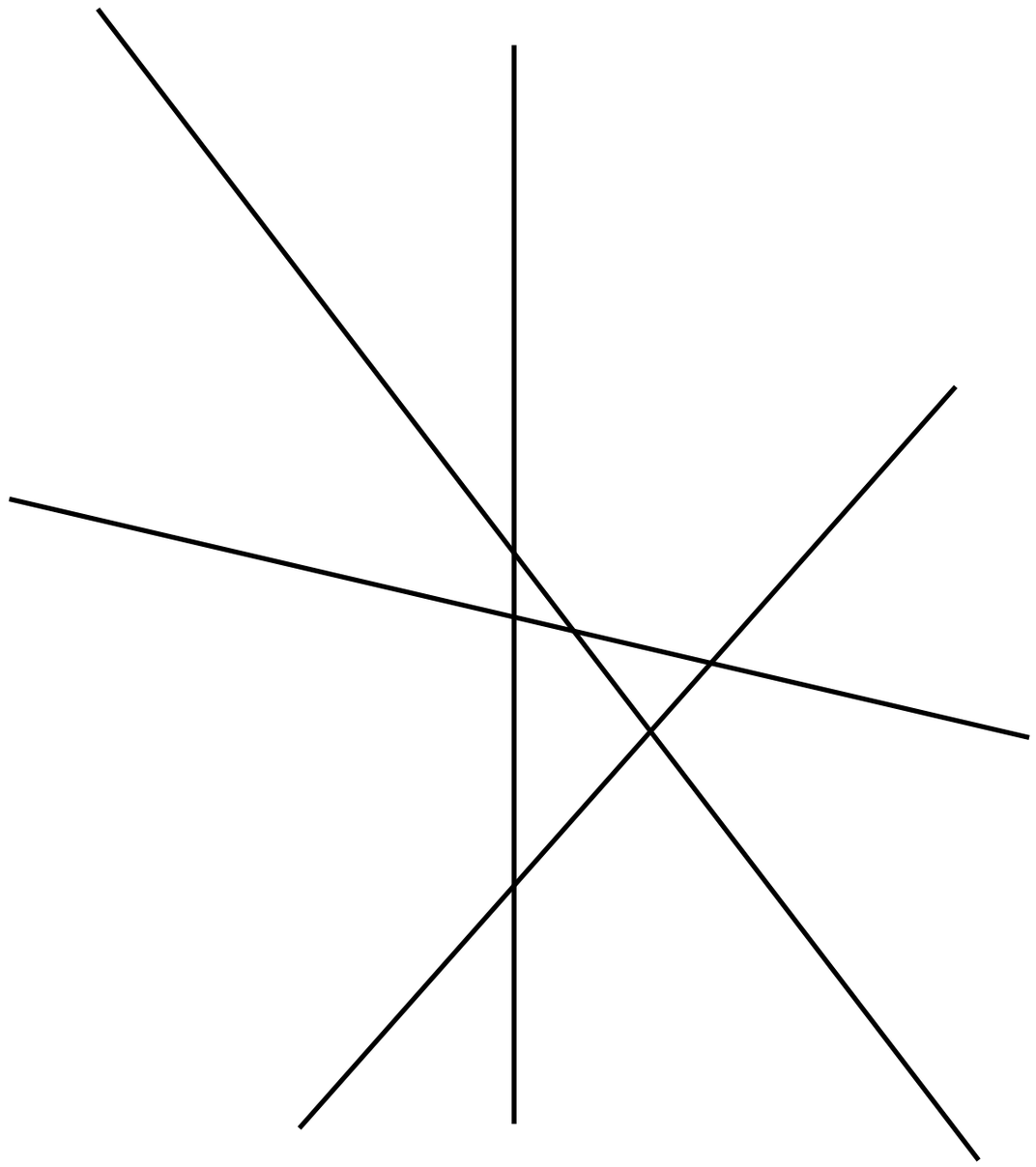
Training and Testing on MNIST

- Obtained 97.4% accuracy with:
  - A single hidden layer of 400 neurons
  - 25 epochs of training

Type	Classifier	Distortion	Preprocessing	Error rate (%)
Linear classifier	Pairwise linear classifier	None	Deskewing	7.6[9]
K-Nearest Neighbors	K-NN with non-linear deformation (P2DHMDM)	None	Shiftable edges	0.52[17]
Boosted Stumps	Product of stumps on Haar features	None	Haar features	0.87[18]
Non-Linear Classifier	40 PCA + quadratic classifier	None	None	3.3[9]
Support vector machine	Virtual SVM, deg-9 poly, 2-pixel jittered	None	Deskewing	0.56[19]
Neural network	2-layer 784-800-10	None	None	1.6[20]
Neural network	2-layer 784-800-10	elastic distortions	None	0.7[20]
Deep neural network	6-layer 784-2500-2000-1500-1000-500-10	elastic distortions	None	0.35[21]
Convolutional neural network	6-layer 784-40-80-500-1000-2000-10	None	Expansion of the training data	0.31[14]
Convolutional neural network	6-layer 784-50-100-500-1000-10-10	None	Expansion of the training data	0.27[15]
Convolutional neural network	Committee of 35 CNNs, 1-20-P-40-P-150-10	elastic distortions	Width normalizations	0.23[8]
Convolutional neural network	Committee of 5 CNNs, 6-layer 784-50-100-500-1000-10-10	None	Expansion of the training data	0.21[16]

Why Do Neural Networks Perform So Well?

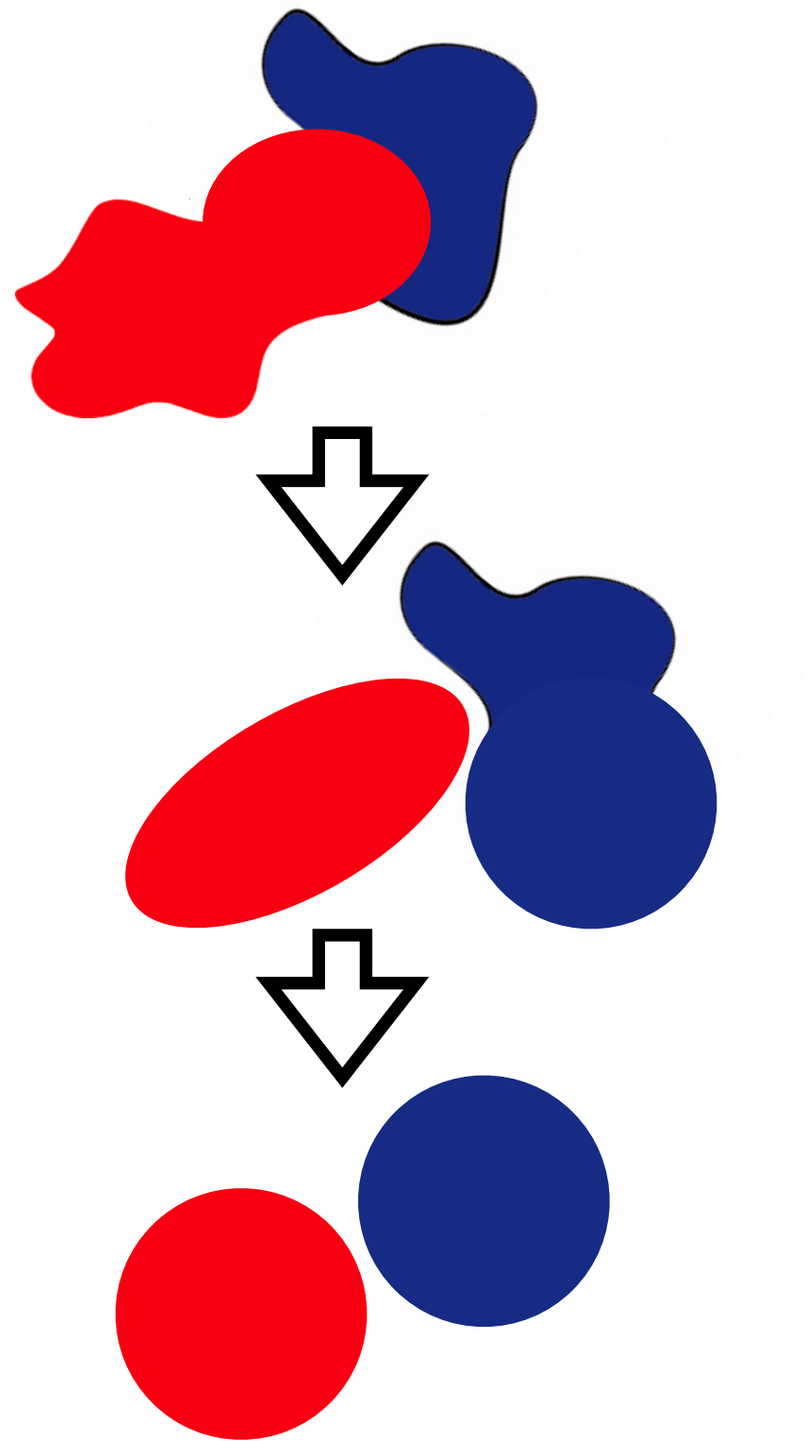
- We can think of each neuron as a simple classifier, dividing the space into two regions:
  - The region in which it fires on the input vectors
  - The region in which it does not
- Then the number of expressible subregions increases exponentially with the number of neurons



Why Do Neural Networks Perform So Well?

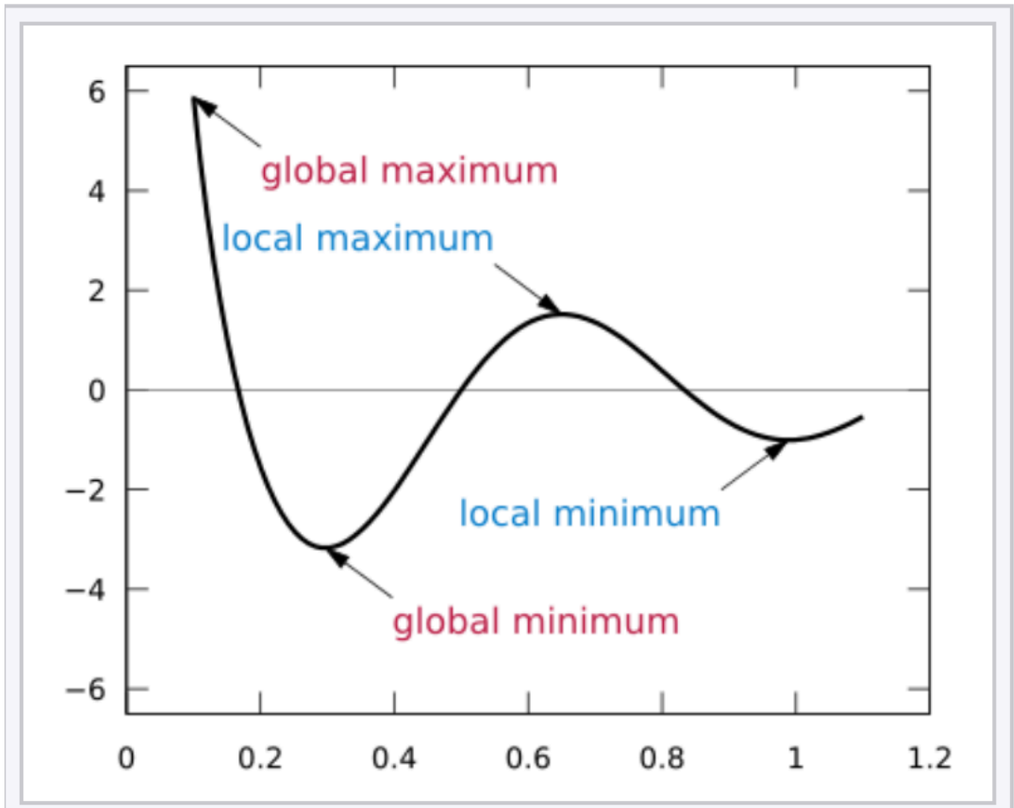
- Each layer transforms the space from the previous layer

- Regions after transformation should be easier to classify



Why Do Neural Networks Perform So Well?

- Traditionally, neural networks were thought to produce unstable results because the gradient might descend into a local minimum



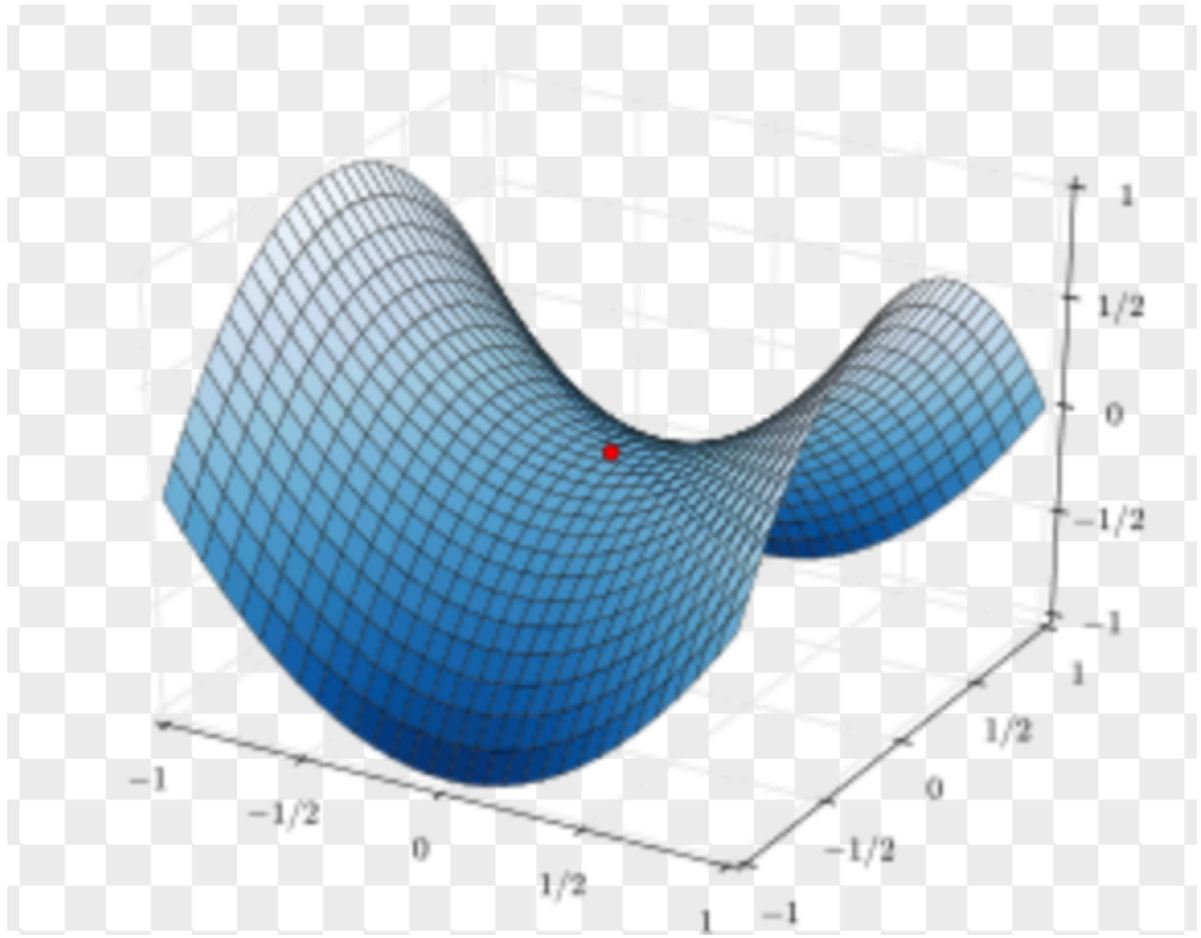
Local and global maxima and minima for $\cos(3\pi x)/x$, $0.1 \leq x \leq 1.1$

Why Do Neural Networks Perform So Well?

- It turns out that, for a surface in a high dimensional space, there are few local minima

- At a given point, along each dimension:

    - Suppose there is a one half chance of being at a local minimum



Why Didn't They Work as Well Twenty Years Ago?

- In large part, they did: The theory was mostly there
- However, computing power was not:
  - Ten cycles of Moore's Law:
    - 1000x speedup in processing power and memory
    - Plus! GPU exploitation, TPU exploitation, SSDs, easier distributed computing, cloud computing
- There are many more massive datasets, commonly available

Next Steps

- Successfully incorporate drop-connect
- Exploit GPUs
- Connect to the parallel collections libraries
- Implement convolutional networks